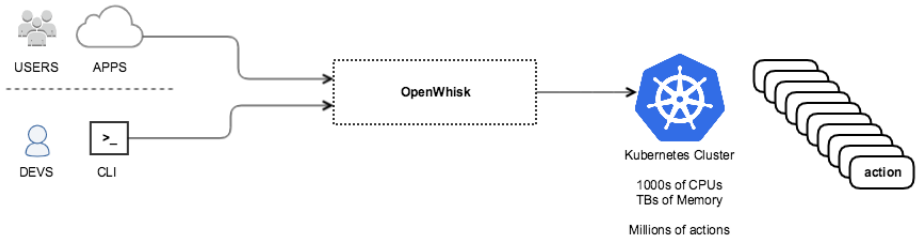
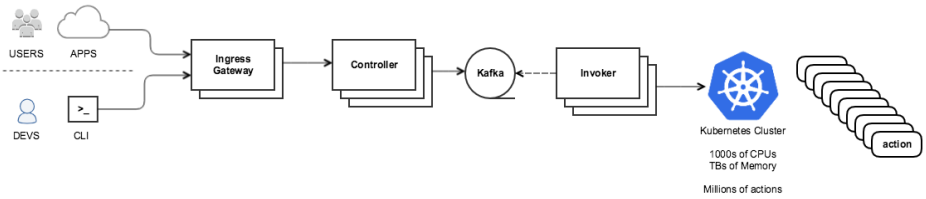
# OpenWhisk on Kubernetes

## Goals




- Try to reuse Kubernetes components in OW
- Keep the existing developer experience and CLI

### Current deployment




Originally OpenWhisk was built with the assumption that each Invoker is responsible for a single VM in the cluster. With a Cluster Manager, this premise changes, as a single Invoker could be in charge of the entire cluster.  The Cluster Manager is responsible for each VM. From the Invoker's perspective, the entire cluster looks like a single pool of resources.

The current OpenWhisk components, Controller and Invoker, have problems managing the same pull of resources. For example:

- when 2 or more Invokers manage the same resources, conflicts may arise due to the fact that Invokers don't share any model
- the load balancing logic in the Controller becomes less important given than it doesn't matter which Invoker executes a given action, because it will still execute it on the same pool of resources
- the max memory limit set per invoker is also not useful

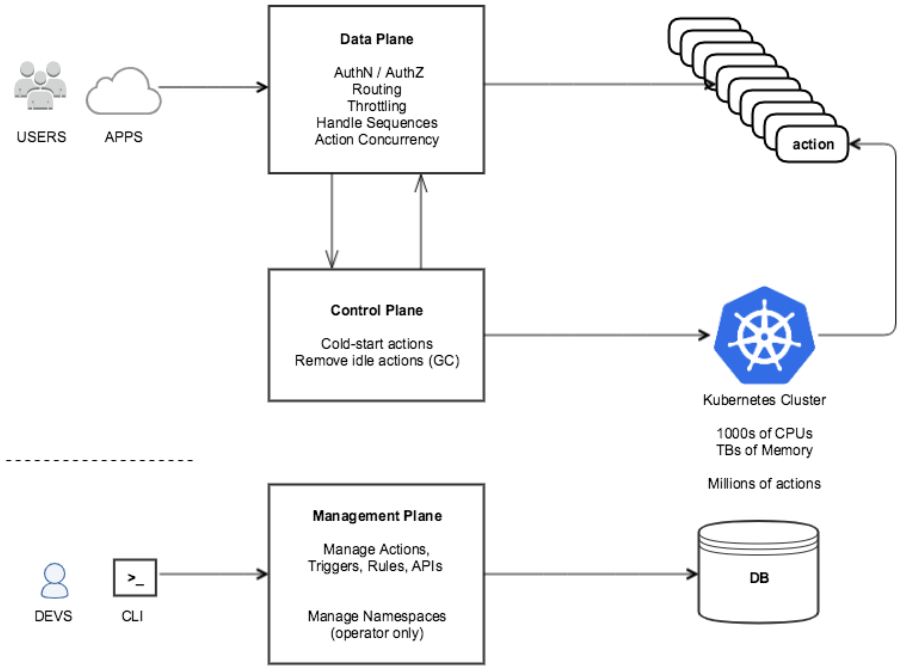
# CNCF based Architecture

Given these new premises, and the experience of building a FaaS solution that the OpenWhisk community has, can the OW system benefit from existing CNCF projects to simplify its implementation, while keeping the same developer experience ?

This document looks at some possible options to achieve this with Kubernetes and other solutions from the CNCF landscape.

## Management, Control, and Data Plane

OpenWhisk system can be decomposed in 3 distinct areas of concern, inspired from Network Devices and Systems designs.




## Management Plane

The management layer exposes an API that is primarily serving developers that manage actions, triggers, rules, and APIs. The `wsk` CLI interacts with this layer.

OpenWhisk operators may interact with this layer to manage namespaces too.

## Control Plane

The control plane layer has 2 main responsibilities: cold-starting new actions, and removing idle actions (Garbage Collector).

This layer could be implemented by reusing the JVM architecture. In a JVM architecture the memory is allocated and is then freed with the help of a GC component. Similarly, a FaaS system needs to allocate resources such as cpu, memory, disk space to each action, then, when an action becomes idle, the FaaS system needs to scale it down to zero, freeing the resources.

The control plane should:

- use the Cluster Manager's API to start and stop actions;
- inform the Data Plane when starting or removing an action;
- provide a configurable GC which should avoid fragmentation where possible; the less fragmentation the more compact the pool of resources is.
- use a mark-and-sweep GC logic to remove containers, to allow enough time for the Data Plane to stop sending traffic to the actions marked for removal

The Control Plane should provide an API used by the Data Plane to cold-start actions, and it should also emit events each time a change in the resource allocation happens; each time GC removes idle containers, or each time a new action is created, the Control Plane should notify all Data Plane instances of such changes.

> To be detailed: Unlike the current OW, the system is "async by default". The new design is "sync by default". The open question is how to handle async cases.

## Data Plane

The data plane layer invokes actions as fast as possible. When an action needs to be cold-started, the data plane delegates this to the Control Plane, awaiting for the action to become ready before invoking it. Once an action is warmed-up the data plane is notified, and if it was waiting for such event in order to invoke an activation, it should resume the execution.

The Data Plane invokes warmed actions without going to the Control Plane. The only time Control Plane is used in an activation flow, is when a cold-start is required.

The Data Plane should stop sending traffic to actions that are marked for removal by the Control Plane. The only exception is when an action marked for removal receives an activation in the mean time, in which case the Data Plane informs the Control Plane, which may choose to remove the "mark for removal" and keep the action running, or recycle the action with a new one.

This layer should have support for sequences, and concurrency of 1 for action invocation.

The Data Plane should perform the Authentication and Authorization that OpenWhisk Controller does currently, and it should decorate the request with the context set by OpenWhisk (i.e. __OW_NAMESPACE, __OW_ACTION_NAME, __OW_ACTIVATION_ID, etc)

# CNCF Projects to integrate with

## Data Plane

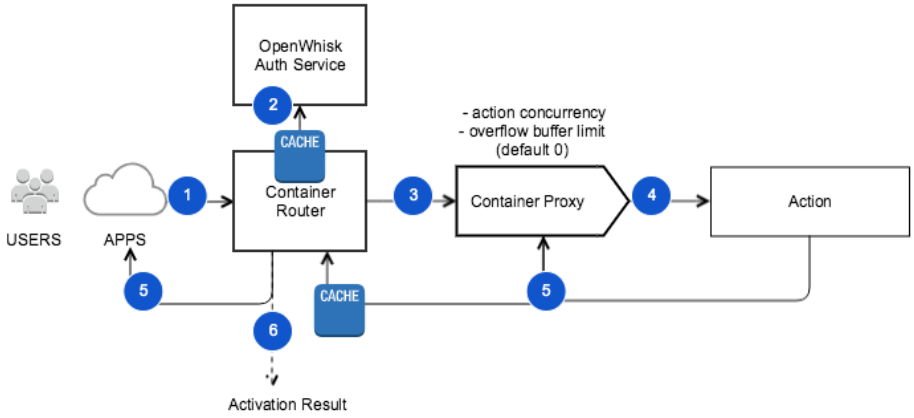
Candidates:

- Envoy blocked URL
- Nginx

Requirements for Data Plane:

- Invoke an authN/authZ service, reusing the existing authN/authZ implementation in OpenWhisk
- Routing
  - Including support for **sequences**
- Throttling
  - Respect namespace limits
  - Respect Action level concurrency
- Caching the response, based on what the action returns. I.e. an action that validates an OAuth token could instruct the system to cache the response for that token until it expires.
- Support API Management
  - Otherwise the existing OpenWhisk Gateway can be reused
- Support for Observability: Metrics, activation info, tracing, etc

### Flow for warm container

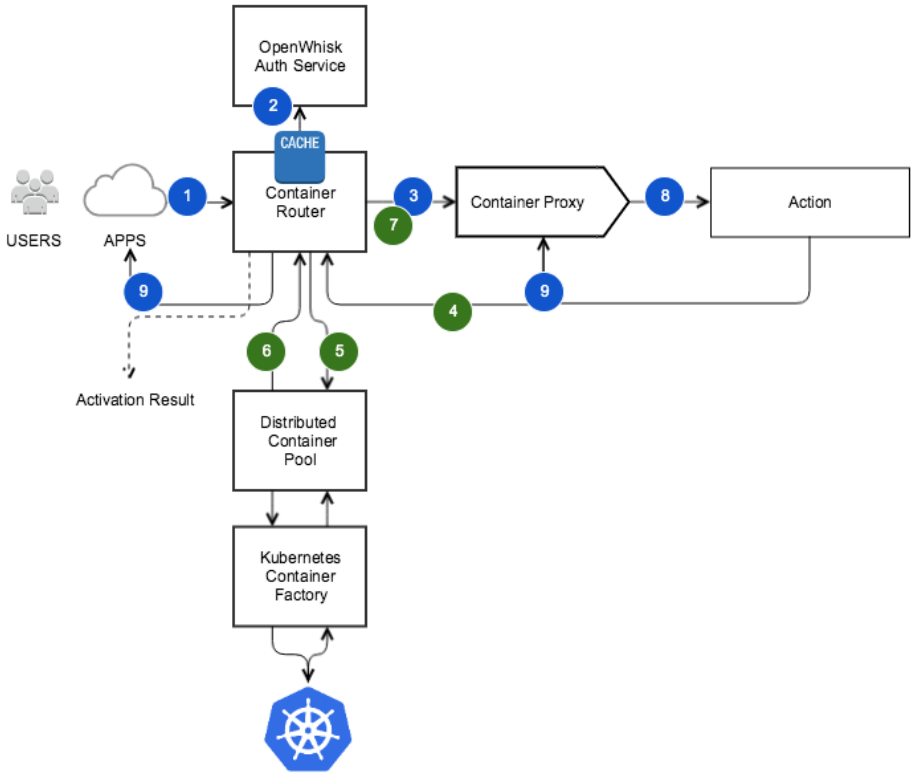



1. The request arrives from a client
2. Authentication and Authorization
   a. The Container Router validates the Authorization header with OpenWhisk Auth Service
   b. The response of the Auth Service is cached
3. Routing
   a. Check namespace limits
   b. Forward the request to a container selected from a list of warmed actions that the Action Router keeps.
      i. **(new)** Streaming the request to the action would be a nice; OpenWhisk doesn't have support for this, and such feature could remove the max payload limits
      ii. **(new)** Websockets could also be supported, another missing feature in OpenWhisk.
4. Container Proxy sidecar
   a. Check action concurrency limit
   b. Buffer a few more requests, queueing them into an overflow buffer; this may be something useful when cold-start could take longer than just queuing a few more requests. Blackbox actions that need to download the docker image may benefit from this more. This idea is inspired from KNative Serving.
5. Invoke the action and return the response
   a. **(new)** Caching the action response could be another nice to have feature, which is not implemented in OpenWhisk. Caching should be controller by the action response.
6. Collect activation info.
7. Sequence support.
   a. If the action is part of a sequence, then the Router should have logic to invoke the next action in the sequence.

i. Other ideas to explore to support sequences, should the support in the ContainerRouter is too difficult to implement
    1. ContainerProxy could "understand" sequences
    2. Or reuse Composer and implement sequence-as-an-action.

### Flow for cold-start

When the Action Proxy is at capacity, it should return a 429 message back to the Container Router. A `Retry-After` header could specify `<delay-seconds>` or `<http-date>` for a CircuitBreaker in the ContainerRouter to avoid routing to that action. The time window for retry should ideally be computed from the response times observed by the Container Proxy.




The green steps are additional steps required for cold-start:

4. Container Proxy returns a `429` indicating the action has reached its max concurrency and can't take more activations. If there's no container running for that action, skip to step 5.

5. Container Router goes to the `DistributedContainerPool` to request a new container to be created

6. After the container is created, all Container Router instances are informed, and the activation proceeds as in the *Flow for the warm container* described above.

## Control Plane

Candidates:

- OpenWhisk Controller and Invoker - refactored into a single service that meets the requirements

Control Plane concerns:

1. Cold-start actions - allocate resource
2. Garbage Collect idle actions - de-allocate resources

The Control Plane should be used by the Data Plane **only** when cold-starting new actions.

### DistributedContainerPool

This Component is at the core of the **Control Plane**. It should be concerned with the following:

- globalPool
    - Cluster Wide view of all running actions
    - Distributed Map with minimum data about actions needed for ResourceAllocator and GC

        - it should sync with Kubernetes from time to time to update the state, in case a container dies, or a Kubernetes operation kills that container
- resourceAllocator - SingletonActor
    - It's in charge to start containers on a node that has resources
    - When allocating resources, Placement Strategies should consider CPU, MEM, GPU, Network, and other resources an action might consume.
- garbageCollector - SingletonActor
    - it removes idle actions
    - It needs to be a singleton so that when deciding what resource to free, in can avoid fragmentation. In other words, it should free resources to make the free space as compact as possible.
        - This is particularly important when scaling down the nodes running actions
    - Its implementation should be configurable and swappable

### Management Plane

This can reuse the OpenWhisk implementation.

Candidates:

- OpenWhisk Controller, slimmed for Management APIs

### Logs

Candidates:

- Fluentd blocked URL

Logs could be captured through fluentd and forwarded to ElasticSearch, Splunk, or other log stores (RDBMS, NoSQL, Hadoop)

The log format for the actions should be updated so that each log line includes activationID, namespace, and other identifiable information needed to serve `wsk api logs <activationID>` command and `wsk api get <activationID>` command.

### Monitoring

Candidates:

- Prometheus blocked URL

OpenWhisk already integrates with Prometheus. See https://github.com/adobe-apiplatform/openwhisk-user-events

### Throttling/Limits

#### Namespace limits

Limits such as max concurrent action containers per namespace could be enforced by the Container Router.

#### Action limits

Limits such as concurrent requests per action container could be enforced by the Container Proxy.

## Previous discussions

Provide support for integration with Kubernetes. One approach could be to deploy and run the components on a Kubernetes provider as we do for Vagrant, Docker, Docker-Compose, and OpenStack.

Proposal to be discussed on the dev list https://lists.apache.org/thread.html/66b2111f8edb4a44466728d697c735f549971909600a02f1b585a9e7@%3Cdev.openwhisk.apache.org%3E